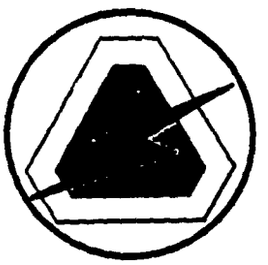
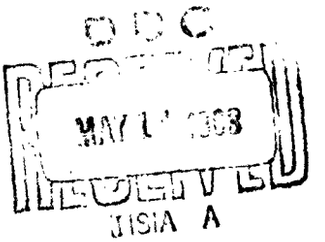
USAELRDL Technical Report 2345

AN APPLICATION OF HEURISTIC PROGRAMMING TO THE PROBLEM OF THEOREM PROVING BY MACHINE

Serafino Amoroso





March 1963

UNITED STATES ARMY
ELECTRONICS RESEARCH AND DEVELOPMENT LABORATORY
FORT MONMOUTH, N.J.

U. S. ARMY ELECTRONICS RESEARCH AND DEVELOPMENT LABORATORY

FORT MONMOUTH, NEW JERSEY

March 1963

USAELRDL Technical Report 2345 has been prepared under the supervision of the Institute for Exploratory Research, and is published for the information and guidance of all concerned. Suggestions or criticisms relative to the form, contents, purpose, or use of this publication should be referred to the Commanding Officer, U. S. Army Electronics Research and Development Laboratory, Attn: Director, Exploratory Research Division "C".

J. M. KIMBROUGH, JR
Colonel, Signal Corps
Commanding

OFFICIAL:
HOWARD W. KILLAM
Major, SigC
Adjutant

DISTRIBUTION:
Special

QUALIFIED REQUESTERS MAY OBTAIN COPIES OF THIS REPORT FROM ASTIA.

THIS REPORT HAS BEEN RELEASED TO THE OFFICE OF TECHNICAL SERVICES, U. S. DEPARTMENT OF COMMERCE, WASHINGTON 25, D. C., FOR SALE TO THE GENERAL PUBLIC.

AN APPLICATION OF HEURISTIC PROGRAMMING TO THE PROBLEM OF THEOREM PROVING BY MACHINE

Serafino Amoroso

DA Task No. 3A99-25-004-03

Abstract

A mechanical procedure using trial and error techniques is outlined which will verify, in a large number of cases, the validity of an argument form expressed in quantification theory. Combinational processes have been used to a minimum extent. Techniques of implementation for a digital computer are also discussed.

U. S. ARMY ELECTRONICS RESEARCH AND DEVELOPMENT LABORATORY

FORT MONMOUTH, NEW JERSEY

CONTENTS

AN APPLICATION OF HEURISTIC PROGRAMMING TO THE PROBLEM OF THEOREM PROVING BY MACHINE

INTRODUCTION

The hope of using "mechanical methods" to achieve significant results in mathematics, such as obtaining mathemathical theorems, seemed on the verge of realization when Hilbert noted that all classical mathematics could be expressed in the language of quantification theory. The ability to determine whether or not a quantificational formula (theorem) follows logically from a finite set of given quantificational formulas (axioms), was the central problem in the hoped for process. This central problem is known now as the Entscheidungs problem. The subsequent work of Turing, Church and others in the 1930's, which showed the unsolvability of this problem resulted in the feeling that "mechanical methods" (which implies our modern computers) could not be used to resolve "significant" problems in mathematics. Renewed interest in this topic has arisen as a result of recently developed methods which will mechanically (effectively) determine that a quantificational argument is valid if and only if it is valid, but will be inconclusive if the quantificational argument is not a logical truth. Also, effective procedures have been developed which will give either an affirmative or negative answer to this question for a significant subset of the set of all quantificational arguments.

There have been a number of attempts to produce a workable machine program, making use of the available procedures, but each has run into considerable difficulty with respect to the propositional (truth-functional) tests required by the methods. Most of the theoretical discussions of the proof procedures, for example, Quine's,[1] point to truth-tables to resolve the problems involving the truth-functional tests. Although this is theoretically an effective process to solve all questions with respect to truth values of propositional formulas for use on computers, this is an impractical approach, since the number of rows on a truth-table increases exponentially with the number of different propositional letters in the given formula under test. To date the best approach to this problem seems to be that of Davis and Putnam.[2] They have presented a procedure for testing truth-functional forms that, with respect to use on a computer, far surpasses the truth-table method, and is more practical than the methods used by Wang[3] and Gilmore.[4] But even the Davis-Putnam [2] test is, in general, quite time consuming, and is still the phase of the overall procedure most in need of improvement. Interesting work on this problem is now being done at IBM by Dr. B. Dunham.

This report is an application of a trial and error technique that simulates the way a human being would attack the problem by extracting additional information from a previous phase of the process to reduce the propositional tests to a point where even the truth-table method might be used. The methods used in this approach (heuristics) do not give rise to a procedure theoretically as powerful as the Davis-Putnam[2] approach. However, improved heuristics may increase the power of this machine procedure, if it is not already sufficiently powerful, to the point where the reduced efforts required in truth-functional testing do warrant the reduced processing power (in the sense that there are certain quantificational arguments that are valid and unprovable by the machine procedure). The question of whether or not the present procedure is too limited has not been adequately investigated.

DISCUSSION

Heuristic Methods in Programming

Practically all problems solved by modern electronic digital computers today have associated with them an effective "algorithm", that is, a systematic procedure which, when presented with the problem as input, is guaranteed to produce a solution as output. There are, however, interesting problems for which no algorithm is known, or no efficient one may be known, or no algorithm is possible. Even though we may not have an algorithm for a given problem, we may at least know how to recognize a solution, as such, should one appear. A minimum requirement associated with problem solving, whether by man or machine, seems to be: If any "solution" should appear, a "method" must exist for deciding whether or not it really is a solution.

Usually we shall have partial information about how to produce a solution to such problems. For example, as a result of successful human attempts at solutions in the past for similar problems, we may have enough information to enable a partial decision procedure. That is, a systematic procedure that can never guarantee solutions to all questions about the problems for which it was designed, but nevertheless, it may be powerful enough to handle successfully a large percentage of problems presented to it. One simply described handling of such problems by machine is to have the machine search and test in a systematic way all possible expressions that could be solutions. Even though we know something about the syntactic form of a solution, such blind, brute force enumeration almost always involves near-astronomical numbers of trails, since these methods usually are little more than the systematic testing of all expressions in some language. Fortunately, it is usually possible to extract added information from the given problem under investigation by the machine, and thereby reduce the number of trial attempts at a solution to a reasonable number.

Before discussing the machine procedure of this report, a detailed discussion of the fundamental notions of quantificational logic needed in the following discussions will be developed.

Definition of Well-Formed Formula in Quantificational Logic

In this section, following the presentations in Patton[5] and Wang,[3] we will present a definition of the syntactical properties of the language of quantificational logic. The elemental symbols that form the alphabet of the language are classified as follows:

Logical connectives: " $\sim$ ", " & ", " $\vee$ ", " $\supset$ ".

Statement letters: " A ", " B ", " C ", ...

Predicate letters: " G ", " H ", " I ", ...

Individual names: " a ", " b ", " c ", ...

Individual variables: " x ", " y ", " z ", ...

Quantifiers: "(Ex)", "(x)", where x is any individual variable.

Since statement letters and predicate letters will always be followed by different syntactical forms (predicate letters will be followed immediately on the right by individual names, individual variables or both, statement letters will never be immediately followed on the right by these symbols), machine procedures do not demand the set of statement letters be mutually exclusive from the set of predicate letters. It will be essential, however, that individual names and individual variables belong to mutually exclusive sets. These points will be discussed in more detail later.

From the set of all possible finite strings of these symbols, a unique subset will be called well-formed formulas (wff's) and will be defined recursively as follows:

1. A statement letter is a wff.

2. A string consisting of a predicate letter followed by any number of individual names and/or individual variables in any combination is a wff, and the variables occurring in it are called free variables.

3. If S is a wff, then so is $\sim$ S (called the negation of S).

4. If S and T are wff's, then so are (S & T), (S V T), and (S $\supset$ T), and the free variables of S and T are also free in these wff's. (They are called, respectively, the conjunction of S and T, the disjunction of S and T, and the implication of T by S.)

5. If S is a wff and x is any variable, then provided "(Ex)" or "(x)" does not occur in S, then (Ex)S and (x)S are wff's, and the variables other than x in S that occurred free still occur free. x in S is now said to be bound by the quantifier.

Interpretation of Quantificational Formulas

Quantificational formulas as such cannot be said to be true or false until an interpretation is presented. An interpretation of a quantificational formula S will consist of the following assignments:

1. To each statement letter in S, a truth value.

2. To the quantifiers of S, a set of individual elements called a universe.

3. To each free variable of S, a member of the universe.

4. To each predicate, with its variables and individual names, a truth value assigned to each string obtained by substituting individual elements from the universe for the variables.

Once an interpretation has been made, the truth value of the quantificational formula is determined by the following recursive rules:

1. A formula without logical connectives or quantifiers obviously has a truth value when its symbols have been interpreted.

2. (S V T) is true if and only if at least one of S or T is true. (S & T) is true if and only if both S and T are true. (S ⊃ T) is false if and only if S is true and T is false. ∼ S is true if and only if S is false.

3. If x is any variable and (x)S is a wff, then (x)S is true for a given universe if and only if S is true under every possible interpretation of x in the given universe.

4. (Ex)S is true in a given universe if and only if S is true for at least one interpretation of x in the given universe.

Since statement letters "disappear" as soon as an interpretation is given, they will not affect any of the problems with which we will be concerned. We shall assume from here on that statement letters do not appear in any formula.

The effect of statement letters in any formual is accounted for according to the following replacement rules:

1. (a) replace ∼ T by F,

   (b) replace ∼ F by T,

2. (a) replace T & B or B & T by B,

   (b) replace F & B or B & F by F,

3. (a) replace T V B or B V T by T,

   (b) replace F V B or B V F by B,

4. (a) replace T ⊃ B by B,

   (b) replace B ⊃ F by ∼ B,

   (c) replace B ⊃ T or F ⊃ B by T.

Definition Based on the Notion of an Interpretation of a Quantificational Formula

1. A non-empty interpretation of a wff is one that assigns to the quantifiers of the wff a non-empty universe. From now on we shall assume when using the term "interpretation" a non-empty one.

2. A wff S is a logical truth if and only if S comes out true on every interpretation.

3. A wff S is consistent if and only if S comes out true on some interpretation.

4. Two wff's S and T are logically equivalent if and only if ((S ⊃ T) & (T ⊃ S)) is a logical truth.

5. A finite set of wff's is consistent if and only if any conjunction of all the wff's is consistent.

6. An argument is valid if and only if the set of wff's that comprises its premise, and the negation of its conclusion, is inconsistent.

Prenex Normal Form

Given any wff S it is always possible to find a wff T which is logically equivalent to S, and T is of the form $(Q_1)\ (Q_2)\ldots(Q_n)(T')$, where (T') contains no quantifiers. Q_i, i = 1, 2, ..., n, is a quantifier (either an (Ex) or (x)). When any wff is in the form of T (that is, all quantifiers on the left), it is said to be in prenex normal form. For a further discussion of this topic see reference 6.

Quine's Proof Procedure for Quantificational Logic

The method to be described now, forms the basis of my machine procedure. It was presented in detail in the original paper by Quine[1] which appeared in the Journal of Symbolic Logic in 1955. The method is a test for the inconsistency of a finite set of quantificational formulas. That is, it can prove any inconsistent set of formulas to be inconsistent, but it cannot prove any consistent set of formulas to be consistent. We can use the method to prove the validity of arguments since an argument is valid if and only if the conjunction of the set of formulas representing the premises of the argument and the negation of the conclusion of the argument is inconsistent. This can be seen more clearly by considering the following: let A be a set of formulas representing the premises of an argument, and let B represent a conclusion. We know that A, and therefore B, is a valid argument if and only if any conjunction of all the premises and the negation of the conclusion is inconsistent. Since, from truth-functional logic, p & $\sim$ q is the negation of p $\supset$ q, then if p & $\sim$q is inconsistent, then p $\supset$ q must be logical truth.

The first requirement of the method is that each formula of the set being tested for inconsistency be put into prenex normal form in such a manner that no variable letter that occurs in any formula as an existential quantifier occurs free anywhere, or as another existential quantifier in the set of formulas. If we now have a set of formulas P_1, P_2, ..., P_n satisfying these conditions, we now construct for each P_i and F_i, called the functional normal form, as follows: delete the leftmost existential quantifier P_i and subscribe $x_1x_2\ldots x_k$ to each variable that it bound, where the universal quantifier that was deleted is $(x_1)\ (x_2)\ldots(x_k)$, in that order. Continue the above process with each next leftmost existential quantifier of P_i finally arriving at the F_i. A term with subscripts in one of the F's is called a function term. What is called the lexicon for the set of P's is a set of elements determined as follows:

1. All variables which occur free and subscriptless in the F's, or if there are none, then the letter a.

2. All results of uniformly replacing the subscript letters of function terms by members of the lexicon.

A lexical instance of F_i results when the quantifiers of F_i are dropped and the variables they bound are replaced by members of the lexicon. The subscript letters of the function terms are to count as bound variables for purposes of this definition. The method is based on the principle that a set of quantificational formulas is inconsistent if and only if some lexical

instance of the set is truth-functionally inconsistent. The completeness and soundness of the method are discussed in references 1, 5, and 7.

The Machine Procedure

The approach we will take in describing the machine procedure for this project will be as follows: we will first describe the mechanical processes involved, in a natural language (English) assuming certain machine processors available to handle the manipulations necessary on the strings of information. The machine capabilities will not always be specifically mentioned in this section. Later we will concentrate on the machine capabilities demanded by our problem.

a. The Machine Formation of the Functional Normal Forms

The procedure for constructing functional normal forms for any finite set of quantificational formulas is completely deterministic and poses little challenge to anyone familiar with the usual processing involved with mechanical string languages. To bring out the "mechanical" features of the following manipulations which will be performed on our information, we shall assume certain registers available and certain operations such as formation and deletion of lists, scanning strings, and concatenation and deconcatenation. All of these, and many others, shall be discussed in detail when we consider the general syntactical mechanical processors made necessary, or just desirable, for a machine realization for our problem. We shall introduce informally each operation and device as needed. Many will not be discussed directly in this section, but will be implied by the required manipulations.

We assume the P_i's are stored in a general storage area and can be called into a general working register A at will. We assume an auxiliary register S which will be used in the manner of a "scratch pad". Both of these and any other we may need will be registers of flexible size. Assume the registers are empty, and bring P_1 into A-register. Scan P_1 from left to right as follows: if the symbols on the extreme left are those of an existential quantifier (left parenthesis, "E", variable name, right parenthesis), then they are deleted. Note here, we are implying the ability to recognize a variable name symbol, therefore, a list must be available of variable names, and a comparison operation is performed. If these first symbols are not those of an existential quantifier, then the machine asks whether or not they are the symbols of a universal quantifier. If they are not, then P_1 is already in functional normal form and is now considered F_1. We store F_1 and call P_2 into the A-register. But, if these symbols are those of a universal quantifier, we do not delete this quantifier, but place a copy of its variable name in auxiliary register S. We then move our scan to the symbols to the right of the quantifier. From now on, every time we meet a universal quantifier in our scan, we concatenate its variable name to the right of the contents of the S-register. Also, from now on, each time we meet an existential quantifier we delete it and concatenate to each variable it bound in the formula, the symbol "/" followed by the contents of the S-register. We do not erase the contents of the S-register during this concatenation operation. The above procedures are continued until something other than a quantifier is met at which time we erase the S-register, store the contents of A in F_1 (the name of the location in general storage containing F_i is "F_i"), and call the contents of P_2 into the A-register. These procedures are continued until all the P's have been transformed into F's.

b. Problems of Lexicon Formation

Many of the processes that we shall consider in this section and especially in the next section on "optimal instantiation" are difficult, if not impossible, to consider in general - let alone solve in general. The best strategy that we have found to attack these problems is to design procedures that may work in a large number of cases, and then through empirical results we can analyze our results and modify our design, or completely rebuild them, if the analysis indicates that such is desirable, or necessary (and of course possible). This is in line with what many researchers, in artificial intelligence systems, consider to be the best strategy for their problems, that is to postulate a system capable perhaps of exhibiting an interesting behavior, then to explore it experimentally and theoretically (modifications being made, of course, as a result of empirical data). Examples of such projects are the neural nerve net experiments of Holland and his associates, the work of Amarel in automatic theory formation processes, and Newell, Shaw, and Simon's work on the Logic Theory Machine and the General Problem Solver.

The system being discussed in this report has undergone some modification. The present section discusses the problems encountered when the system was designed to enumerate a finite number of lexicon members. The number depended upon the "complexity" of the set of P's. Although this process is no longer designed into the system, we feel it advantageous to discuss it here so as to provide additional insight into the programming problems involved, to provide an additional motivation for the processes presently chosen in the system, and finally to illustrate further how blind enumeration, which often gets out of hand, can be replaced by strategic "short-cuts" seen as a result of patterns in the formulas.

The lexicon given in the introduction contains all unbound, subscriptless variables. This was easily programmed: the F's were placed in the A-register and each quantifier letter (all of course are universal) is placed on a previously empty list Q. During a left to right scan of the formula proper, each variable name is tested against the entries on Q, so that if no comparison is found, and the variable under test is not followed immediately by "/", then it is placed on a list L which represnts the lexicon. This process continued through all the F's gives all lexical member names obtained from unbound, subscriptless variables. After testing all the F's, if list L is empty, "x" is placed on list L, in line with the theory of our lexicon. The recursive process involved in extending the lexicon is to now replace all subscript letters by members of the lexicon. This process can generate a representive lexicon, that is, one that is complete enough to solve a large percentage of problems, only if all subscript strings are of length 1 (with perhaps only one of length 2). When an attempt is made to generate a lexicon for a set of formulas in which appear subscript strings of lengths 2, 3, or more, the number of elements needed in the lexicon becomes extremely large. This not only makes the generation of the lexicon difficult, but makes the rest of the machine procedure very inefficient.

As an example of the lexicon difficulties consider the example of the two function terms H_{xyz} and G_{uvwz}. Suppose "a" was previously determined to be a member of the lexicon, therefore, X_{aa}, and u_{aaa}, are members of the lexicon. The following are also members: $x_{x_{aa}a}$, $x_{ax_{aa}}$, $u_{aax_{aa}}$, ..., $x_{au_{aaa}}$, ...

If one makes the attempt, a method of enumeration is most difficult, if not almost impossible, to program in general. Imagine the difficulty of four or five multisubscripted predicates!

c. Toward Optimal Universal Instantiation

The purpose of an attempt toward optimal instantiation of the functional normal forms is to instantiate in such a way so as to get all strings of lexical members following like predicates to be identical. This is desirable since, the more identical lexical instance terms we have in a lexical instantiation of the set of F's, the greater will be our chance of arriving at a truth-functional contradiction which is our overall goal. This phase of the machine procedure proceeds as follows: a scan is made of the functional normal forms and lists are formed, each containing like predicates followed by the strings of symbols that appeared after each predicate. During the formation of the lists a parenthesis is placed on each side of a symbol that contains a bound variable (all subscripted terms, and bound unsubscripted terms). Along with each entry in the lists is placed its location in the set of F's, that is, the particular F from which the predicate and string was extracted, and the i^{th} occurrence of that particular predicate taken from left to right within the F. For example, "$G(x)(u_x)w$, F_2, 3" would be typical, and would mean the third occurrence of predicate G within F_2, containing the unbound variable w, the bound variable x, and the subscripted term u_x. These lists are numbered 1 to m, where m is the number of predicate letters that appear in the set of F's.

A copy of lists 1 through m is made. The reason for this is that instantiations will be made and the results tested: if they fail to produce a contradiction in the final (truth-functional) test, a different instantiation will be made and again tested. We must have a master copy of the lists from which we can make copies for use in our attempts at instantiation.

A copy of list 1 is considered first with an examination of the first symbol following each predicate letter. An attempt is now to be made to maximize the number of identical symbols we can substitute into this position by lexical instantiation. Obviously, if more than one unbound variable type appears in this first position after the predicate letters, these can never be made identical. If, however, at least one unbound variable appears along with subscripted terms, then a decision must be made as to the choice of instantiation. If the number of subscriptless, bound variables plus one is greater than the number of like letters followed by subscript strings of equal length (e.g., u_{xyz}, u_{zwv}, u_{wxv}) plus the number of subscriptless, bound variables, then the subscriptless, bound variables and parentheses are placed by the unbound variable. On the other hand, if this is not the case (the number of unbound, subscriptless variables plus 1 is not greater), then the subscripted letters in the terms which qualified are instantiated with the unbound variable, parentheses are dropped, and the bounded, subscriptless variables are each instantiated with a representative subscripted term which has just had its subscripts instantiated with the bound variable. After all this, if any first symbols following the predicate letters have not yet been instantiated, the process is repeated starting with the copy of list 1 as it is now (after the partial instantiation).

At this point we have instantiated the first symbol place following all predicate letters on list 1. Before going to the second symbol position, we must consider the effects of our instantiations made in symbol position 1. Whenever an instantiation is made in a quantificational formula, every occurrence of the variable within the scope of the quantifier must be instantiated in the same way. Each time, therefore, an instantiation is made in a particular F, it must be similarly made for each occurrence of the bound variable in each similar F in all lists. This, of course, is done by means of the location information following the predicate strings on all lists.

We are now in a position to examine position 2 (still on list 1) for purposes of instantiation. The procedures will be identical with those discussed above for position 1 except that now we have an added criterion. Since the symbols in position 1 have been partitioned into sections of identical lexicon members, we now wish to instantiate in position 2 as to match the partition of position 1 as closely as possible. The procedure is as follows: instantiations of position 2 are made as outlined for position 1 and each result stored. That is, the partitions of position 2 may be different, and if so, each is stored. Each partition (instantiation pattern) is then compared with the partitioning of position 1. The one that matches best is chosen as the instantiation of position 2. The procedure then continues with position 3, which is compared with position 2, and so forth. (Better still, but still not optimal, is that position n ought to be compared with positions 1, 2, ..., n-1, and maximized for match, but for simplicity, and since it does not seem to have an important effect, this will not be done here.) Each time an instantiation is chosen, then each occurrence of the bound variable, again, is similarly instantiated if occurring within the same F.

When all instantiations on list 1 have been determined, we proceed in an identical fashion with list 2, then list 3, etc. When this process is complete we are ready for the next phase of the overall machine procedure, namely, the test for a truth-functional contradiction.

The truth-functional forms are built from a copy of the set of F's without quantifiers and with the instantiations as now appear on the lists of predicate types. The test procedure for the truth-functional contradiction will be discussed in the next section. If the truth-functional test at this point assures us of a contradiction, our machine procedure concludes that the original argument was valid. If, however, we are unable to find a contradiction, we must make an appropriate modification and a new attempt at forcing a truth-functional contradiction. The motivation for the renewed attempt procedures we have chosen is given after its procedures are described.

We renumber the previously mentioned lists of predicates as follows: 1 becomes 2, 2 becomes 3, ..., m-1 becomes m, and m becomes 1. Then the entire procedure for instantiation and the truth-functional test is repeated on a new copy of the original predicate lists. If the condition occurs that the list originally numbered 1 progresses to the point where it is now numbered m and still the truth-functional test fails, then the machine procedure is unable to conclude anything about the original quantificational argument and the machine halts.

The reasoning behind the above change in the list numbering, as a basis for our renewed attempts to derive a truth-functional contradiction, can be

seen from the following example: suppose the following lists had been originally formed by the scan of some F's:

List 1 $Pw, F_1, 1$

$P(x), F_2, 1$

$P(u_x), F_3, 1$

List 2 $H(x), F_2, 1$

$H(y_x), F_2, 2$

Our procedure would instantiate P(x) on list 1 line 2 as Pw, since our procedure would instantiate Pu_x for P(x) only if there would be a resulting gain as far as the number of like predicate strings is concerned, which in this case does not happen. This instantiation of w for (x) forces H(x) of list 2 line 1 to be instantiated by Hw. This would make it impossible to instantiate $H(y_x)$ so as to be identical with Hw. Suppose further that no contradiction could be found in the truth-functional instances unless the two predicates on list 2 are instantiated with the same lexical member. Thus, unless we instantiate list 2 first, no contradiction can be found, and although the original argument is valid we will not find it so by not extending our instantiation procedure. Now, by instantiating list 2 first: Hy_x, Hy_x, Pw, Py_x, Pu_x will be obtained and will result in our machine procedure being able to state that the original argument is valid.

d. Testing for Truth-Functional Inconsistency

After a lexical instantiation of the functional normal forms, they are no longer a set of quantificational formulas, but are then a set of truth-functional formulas. At this stage of the overall procedure, the set must be tested for a contradiction. That is, the question must be answered as to whether or not it is possible to assign values of true or false to the "truth-functional atoms" (the smallest grouping of letters that can take on a truth-value in the set of formulas, i.e., "Gu_{aa}", or "$Hy_{x_a}ax_a$", etc.) so as to get all formulas of the set at once, that is, the same truth-value to all occurrences of like symbols throughout the set of formulas. If the answer is no, that is, no such assignment is possible, then the set is said to be truth-functionally inconsistent, and our procedure assures us that the original argument presented to the machine for test is valid.

Questions such as the one above, concerned with whether or not a set of truth-functional formulas is contradictory, consistent, inconsistent, etc., are all decidable, since truth-tables can always, in theory, be used in a machine decision procedure. There are many machine procedures that have been designed to answer such questions about truth-functional formulas. Each designed to be far more efficient than a brute force table-building procedure. (See for example reference 8.)

Rather than attempt to design a different method of testing for truth-functional inconsistency, which would probably not be as good as the above mentioned works, we will assume that one of the existing methods has been adapted to our needs.

A Summary of the Syntactical Devices Required for an Efficient Machine Realization

In our discussions of the processes that are involved in the machine attack on the problem, certain processing devices and capabilities were assumed to exist, i.e., scanning registers, forming and interrogating lists of information, etc. None of the processes used are beyond the capabilities of existing machines or some programming languages in today's technology. We will here summarize informally certain functions that would be desirable to any machine or language given the job of realizing the overall machine procedure of this thesis.

The machine must have a list of variable names, and a list of individual names, both stored in the machine before any given problem can be worked on. The lists must be mutually exclusive. The machine may also have a list of predicate letters, and a list of sentential letters, and if these were also mutually exclusive lists it would simplify the procedure somewhat. This is not absolutely necessary, however. If the machine can, as a minimum, recognize capital letters, and if a capital letter is followed by a variable name or an individual name, then it is a predicate letter, otherwise it is a sentential letter.

The machine* will have a general storage area that will store strings of symbols, the strings being of variable length, and lists of information (e.g., predicate letters and strings of symbols) of variable depth. The strings and lists will be stored with an addressing capability as follows: a string named, e.g., F_2 will be stored in a location also called "F_2". Therefore, the machine can "store string P_3," or "put string F_1 into the A-register", etc. The machine will also handle lists in a similar way. We can say, for example, "store list 1", or "interrogate list 3 for the following...". Another important machine function will be the ability to duplicate a string or a list. This capability was used throughout our program, and seems to be an important function in problems of this nature.**

The machine will have a general working register that will be able to hold a string of symbols. The register will be flexible in the sense of being able to hold strings of variable length. Parts of the string held in this register can be replaced by strings which need not be the same size as the parts they are replacing. Therefore, the result of such a replacement may be a string in this working register of equal, smaller, or longer length. The register will be composed of cells which will hold basic symbols of our system, e.g., a cell might hold a predicate letter, a "/", a variable name, a "(", etc. These cells will be interrogated usually by a left to right scan, and the machine will react as a function of the symbol found in the particular

* We will now be referring to an "ideal machine (program or hardware) for the mechanization of our procedures.

** It is obviously connected with trial and error techniques.

cell under interrogation. We will usually use this capability as follows: we want to ask if some string X is the head of the contents of the working register (i.e., if X concatenated with Y is the contents of the working register). X might be of the form "(", concatenated with a general name of some type of symbol, followed by ")". The machine will be aided in situations of this sort by table look-up procedures to determine whether symbols are of such and such type, or of such and such a class.

The machine is able to concatenate onto, or deconcatenate from, either end of this working register. The machine can store the contents of the working register into general storage, clearing, or not clearing itself, as it does so. We can call strings from general storage into the working register, each such operation will clear the previous contents, if any, of the working register.

The machine also has an auxiliary register that can be cleared, strings can be copied from it, and strings can be concatenated onto either end of its contents. It must also be capable of holding strings of variable lengths. Frequently used capabilities will be: deconcatenate a string X from the working register and concatenate to the right of the contents of the auxiliary register. Or, replace by the contents of the auxiliary register, each occurrence in the string contained in the working register, of the symbol y.

With regard to the realization of our machine procedure on a computer, symbol manipulation languages are well-suited to problems of this nature. Comit, a new user-oriented general purpose symbol manipulating programming language[9] seems to be the one best suited to our problem. The language was designed to make operations on strings of information extremely easy to perform, and very natural to use. When we spoke of placing strings of arbitrary length, and lists of arbitrary depth into a general storage area, this storage area in Comit would be the "shelves" of the language. The general working register of which we spoke would be Comit's "workspace". The auxiliary register that we needed could be any shelf, since, among other operations, Comit allows concatenation or deconcatenation on the left or right of any shelved string. Scanning a string in the workspace of Comit and performing insertions or deletions is the most powerful feature of the Comit system, since it gives one a feeling of naturalness in not having to worry about such things as allocation, overflow, register size, etc. Strings, perhaps representing lists, can be duplicated easily within the system. The Comit system is highly recommended to anyone interested in machine solutions for problems which require programming techniques similar to those discussed.

An Illustration of the Machine Operating on a Problem*

Given:

P_1 $(x)(Ey)((Fx. \sim Gx) \supset Hxy.Jy))$

P_2 $(Ew)(x)((Kw.Fw. (Hwx \supset Kx))$

P_3 $(x)\ (Kx \supset \sim Gx)$

P_4 $(x) \sim (Kx.Jx)$ This last line is the negation of the conclusion which follows from the first three lines.

*The argument used in the illustration is from Quine, reference 1.

First Phase: The Machine Formation of the Functional Normal Forms

	A-Register	S-Register
Time 1	(empty)	(empty)
Time 2	$(x)(Ey)((Fx.\sim Gx) \supset (Hxy.Jy))$	(empty)
Time 3	$(x)(Ey)((Fx.\sim Gx) \supset (Hxy.jy))$	x
Time 4	$(x)((Fx.\sim Gx) \supset (Hxy_x.Jy_x))$	x
Time 5	(empty)	(empty)

The contents of the A-Register were transferred to storage location named "F_1".

Time 6 ... The above process continues for P_2, P_3, and P_4.

Second Phase: The Machine Instantiation

The F's determined from Phase 1 are:

F_1 $(x)((Fx.\sim Gx) \supset (Hxt .Jy))$

F_2 $(x)(Kv.Fw(Hwx \supset Kx))$

F_3 $(x)(Kx \supset \sim Gx)$

F_4 $(x) \sim (Kx.Jx)$

The machine proceeds as outlined on pages 7 and 8 forming the following lists:

List 1	$F(x)$, F_1, 1
	$F w$, F_2, 1
List 2	$G(x)$, F_1, 1
	$G(x)$, F_3, 1
List 3	$H(x)(y_x)$, F_1, 1
	$H w (x)$, F_2, 1
List 4	$J(y_x)$, F_1, 1
	$J(x)$, F_4, 1
List 5	$K w$, F_2, 1
	$K(x)$, F_2, 2
	$K(x)$, F_3, 1
	$K(x)$, F_4, 1

The first instantiation attempt evolves as follows:

List 1	Fw, F_1, 1	(x) was instantiated as w, therefore, each occurrence of the bound variable x in formula F_1 is determined as w.
	Fw, F_2, 1	
List 2	Gw, F_1, 1	
	G(x), F_3, 1	
List 3	Hw (y_w), F_2, 1	
List 4	J y_w, F_1, 1	
	J(x), F_4, 1	
List 5	K w, F_2, 1	
	K(x), F_2, 2	
	K(x), F_3, 1	
	K(x), F_4, 1	

Continuing the process:

List 1	F w, F_1, 1	
	F w, F_2, 1	
List 2	G w, F_1, 1	
	G w, F_3, 1	(x) was instantiated as w, therefore, each bound x in formula F_3 is determined as w.
List 3	H w(y_w), F_1, 1	
	H w(x), F_2, 1	
List 4	J y_w, F_1, 1	
	J(x), F_4, 1	
List 5	K w, F_2, 1	
	K(x), F_2, 2	
	K w, F_3, 1	
	K(x), F_4, 1	

Continuing the process:

List 1 F w, F_1, 1

F w, F_2, 1

List 2 G w, F_1, 1

G w, F_2, 1

List 3 H w y_w, F_1, 1

H w y_w, F_2, 1 (x) was instantiated as y_w for (x) in list 4 gives rise to the following truth-functional forms:

1. $(Fw \,.\, \sim Gw) \supset (Hwy_w . Jy_w)$
2. $Kw \,.\, Fw\ (Hwy_w \supset Ky_w)$
3. $Kw \supset \sim Gw$
4. $\sim (Ky_w \,.\, Jy_w)$

Since Kw and Fw must be true (from 2) then Gw must be false (from 3). Hwy_w and Jy_w must therefore be true since $(Fw. \sim Gw)$ is true (from 1). Ky_w must be true since Hwy_w is true (from 2) Jy_w must be false, since Ky_w is true (from 4). But Jy_w cannot be true and false, therefore the set of truth-functional forms is contradictory. And, therefore, the original argument:

$(x)(Ey)((Fx. \sim Gx) \supset (Hxy.Jy))$

$(Ex)(x)((Kw.Fw.(Hwx \supset Kx))$

$(x)(Kx \supset \sim Gx)$

therefore, $(Ex)(Kx.Jw)$ is valid.

Problems for Future Research

How can one compare the problem-solving power of the machine procedure of this report with other known methods?

If the machine fails to find a proof, how can further attempts be made by an extended machine to increase the overall power of the procedure?

Give a proof that the main heuristic, namely making all strings following like predicate letters identical, is the best goal with respect to forcing a truth-functional contradiction.

Is the method used in the attempt at getting the predicate strings identical the best one?

REFERENCES

1. Quine, W. V., "A Proof Procedure for Quantication Theory," Journal of the Association for Symbolic Logic, 1955.

2. Davis, M., and Putnam, H., "A Computing Procedure for Quantification Theory," Association for Computing Machinery (Journal of), 1960, July.

3. Wang, H., "Toward Mechanical Mathematics," I. B. M. Journal of Research and Development, January 1960.

4. Gilmore, P., "A Proof Method for Quantification Theory," I. B. M. Journal of Research and Development, January 1960.

5. Patton, T.,"Lecture Notes for Philosophy 524," University of Pennsylvania, 1961-1962.

6. Kleene, S., "Introduction to Metamathematics," Van Nostrand, 1952.

7. Patton, T., "A System of Quantificational Deduction," Philosophy Department, University of Pennsylvania, 1962.

8. Copi, I., "Programming an Idealized General Purpose Computer to Decide Questions of Truth and Falsity," University of Michigan, 1959.

9. Yngve, V., "Introduction to Comit Programming," M. I. T., 1961.

DISTRIBUTION LIST

	Copies
Commanding General U. S. Army Electronics Command ATTN: AMSEL-AD Fort Monmouth, New Jersey	3
Office of the Assistant Secretary of Defense (Research and Engineering) ATTN: Technical Library Room 3E1065, The Pentagon Washington 25, D. C.	1
Chief of Research and Development Department of the Army Washington 25, D. C.	2
Chief, United States Army Security Agency ATTN: ACofS, G4 (Technical Library) Arlington Hall Station Arlington 12, Virginia	1
Commanding Officer U. S. Army Electronics Research and Development Activity ATTN: Technical Library Fort Huachuca, Arizona	1
Commanding Officer U. S. Army Electronics Research and Development Activity ATTN: SELWS-AJ White Sands, New Mexico	1
Commanding Officer U. S. Army Electronics Research Unit P. O. Box 205 Mountain View, California	1
Commanding Officer U. S. Army Electronics Materiel Support Agency ATTN: SELMS-ADJ Fort Monmouth, New Jersey	1
Commanding General U. S. Army Satellite Communications Agency ATTN: Technical Documents Center Fort Monmouth, New Jersey	1
Commanding Officer U. S. Army Engineer Research and Development Laboratories ATTN: Technical Documents Center Fort Belvoir, Virginia	1
Commanding Officer U. S. Army Chemical Warfare Laboratories ATTN: Technical Library, Building 330 Army Chemical Center, Maryland	1
Commanding Officer Harry Diamond Laboratories ATTN: Library, Building 92, Room 211 Washington 25, D. C.	1
Headquarters, United States Air Force ATTN: AFCIN Washington 25, D. C.	2
Rome Air Development Center ATTN: RAALD Griffiss Air Force Base New York	1
Headquarters Ground Electronics Engineering Installation Agency ATTN: ROZMEL Griffiss Air Force Base New York	1
Commanding General U. S. Army Materiel Command ATTN: R&D Directorate Washington 25, D. C.	2

Distribution List (Cont)

	Copies
Aeronautical Systems Division ATTN: ASAPRL Wright-Patterson Air Force Base Ohio	1
U. S. Air Force Security Service ATTN: ESD San Antonio, Texas	1
Headquarters Strategic Air Command ATTN: DOCE Offutt Air Force Base, Nebraska	1
Headquarters Research & Technology Division ATTN: RTH Bolling Air Force Base Washington 25, D. C.	1
Air Proving Ground Center ATTN: PGAPI Eglin Air Force Base, Florida	1
Air Force Cambridge Research Laboratories ATTN: CRXL-R L. G. Hanscom Field Bedford, Massachusetts	2
Headquarters Electronic Systems Division ATTN: ESAT L. G. Hanscom Field Bedford, Massachusetts	2
AFSC Scientific/Technical Liaison Office U. S. Naval Air Development Center Johnsville, Pa.	1
Chief of Naval Research ATTN: Code 427 Department of the Navy Washington 25, D. C.	1
Bureau of Ships Technical Library ATTN: Code 312 Main Navy Building, Room 1528 Washington 25, D. C.	1 (2)
Chief, Bureau of Ships ATTN: Code 454 Department of the Navy Washington 25, D. C.	1
Chief, Bureau of Ships ATTN: Code 686B Department of the Navy Washington 25, D. C.	1
Director U. S. Naval Research Laboratory ATTN: Code 2027 Washington 25, D. C.	1
Commanding Officer & Director U. S. Navy Electronics Laboratory ATTN: Library San Diego 52, California	1
Commander U. S. Naval Ordnance Laboratory White Oak Silver Spring 19, Maryland	1
Commander Armed Services Technical Information Agency ATTN: TISIA Arlington Hall Station Arlington 12, Virginia	20
USAELRDL Liaison Officer U. S. Army Tank-Automotive Center Detroit Arsenal Center Line, Michigan	1
USAELRDL Liaison Officer Naval Research Laboratory ATTN: Code 1071 Washington 25, D. C.	1
USAELRDL Liaison Officer Massachusetts Institute of Technology Building 26, Room 131 77 Massachusetts Avenue Cambridge 39, Massachusetts	1

Distribution List (Cont)

	Copies
USAELRDL Liaison Office Aeronautical Systems Division ATTN: ASDL-9 Wright-Patterson Air Force Base Ohio	1
U. S. Army Research Liaison Office Lincoln Laboratory P. O. Box 73 Lexington, Massachusetts	1
USAELRDL Liaison Officer Rome Air Development Center ATTN: RAOL Griffiss Air Force Base New York	1
USAELRDL Liaison Officer U. S. Army Combat Developments Command, CDCLN-EL Fort Belvoir, Virginia	1
USAEMSA Liaison Engineer USASCAJ APO 343 San Francisco, California	1
Technical Dir., SELRA/CS Headquarters, USAELRDL	1
USAELRDA-White Sands Liaison Office SELRA/LNW, USAELRDL	1
AFSC Scientific/Technical Liaison Office SELRA/LNA, USAELRDL	1
Corps of Engineers Liaison Office SELRA/LNE, USAELRDL	1
Marine Corps Liaison Office SELRA/LNR, USAELRDL	1
USACDC Liaison Office SELRA/LNF, USAELRDL	2

	Copies
Chief, Technical Information Division Headquarters, USAELRDL	6
USAELRDL Technical Documents Center SEIRA/ADT, Hexagon	1
Commanding Officer U. S. Army Security Agency Processing Center Fort Monmouth, N. J.	1
Chief Scientist U. S. Army Electronics Command ATTN: AMSEL-SC Fort Monmouth, N. J.	1
File Unit Nr. 1 Rm. 3D-116, Hexagon	1
Director of Research SEIRA/DR, USAEIRDL	1
Director, Institute for Exploratory Research, USAEIRDL	1
Technical Staff, IER	30
Exploratory Research Div. C	10

Army-Ft Monmouth,NJ-MCN 1840-63